目录

可替代团队领袖培养计划

> 对于工作上应该做什么事，如果你没有自己的想法，而是完全听主管的，很危险。多数的主管不会在乎你的成长，也不会把公司的利益摆第一位，他们最在乎的是自己的工作绩效，而他们的工作绩效是要靠你们去达成的。完全听他们的任务布置去做，没有自己的主张，东一榔头，西一棒槌，几年下来就变打杂的了。 ——蔡学镛

本书主要包括如下几个方面：

一、系统化设计

包括产品设计和架构设计。

二、项目管理

实现开发过程中的一些技术、管理、流程小细节。

三、系统运维

包括自动化测试、持续集成（CI）等。

设计

思维导图

工具：

- Mindjet MindManager

UML 建模

流程图、时序图、用例图等，为基本技能。

工具：

- Microsoft Visio （Win）
- OmniGraffle （Mac）

原型

工具：

- Axure （必备）
- Pencil
- Balsamiq Mockups
- Sketch

规范

设计内容上的优劣，需要细看、深思才能知道。但规范上的问题，第一眼，就能发现。

所以输出的图表、文档都要规范。这是最基本要求。

以流程图为例：

- 必须有开始、结束，有且只有一个开始

- 箭头必须画在流程线的尖部
- 只有判断有两个分支流向，其余均为一个

等等其他细节也须注意。

需求

需求收集与整理，可以多画画思维导图，因为部分需求之间是存在关联关系的，要将需求的关系、层级理顺。

什么样的需求该忽略

没有大量数据证明切实符合用户实际需要的。

1.拍拍脑袋得来的想法，往往是没用的

硬币皆有两面，要用批判的眼光去审视产品经理的点子。

当下这个阶段，人人都是产品经理，各种野路子如雨后春笋，导致好的产品经理凤毛麟角。

当产品经理拍拍脑袋提出了一个想法，该做的事情是，让他先去做一个详细的市场调研，给出报告和可行性分析。

举一个我见过的例子：

很早之前我们团队接到一个任务，设计一款新的网关产品。产品经理的想法是将受众用户群体定位在青年人。这样就跟小米——“为发烧而生”不谋而合，直面迎来了一个还算比较强大的竞争对手。

当时我提出了一个针对老龄化的设想，主题是傻瓜化、真智能，让中老年人都能轻松上手的产品。直到2016年，才出了“爱国者聚路由”这样稍微有点神似的产品。

2.用户反馈的信息，不应该直接纳入需求

根据二八原则，将80%精力放在20%最有价值产出的事情上。

用户的需求是需求，但不一定是大众需求。所以如果是一个只有三五十活跃用户的反馈组里，得到的反馈信息仅能作为参考。

举个例子：

假设有这样一个问题：智能门锁通过手机解锁是否需要输入密码。在用户群里，有一些用户反馈说手机App上开锁还要打密码很麻烦，不如去掉这一步的密码校验，得到了一批人的支持。

但这样的需求不可取。实际的需求依然需要大量的数据去支撑。一方面，方便和安全，都需要考虑。另一方面，如果大量的用户反馈觉得这样比较麻烦，最佳实践应该是，保留App上的密码解锁功能，但可以设置开启或关闭，默认开启，由用户去控制，为了方便可以将其关闭，但由于这种用户自发行为导致的安全隐患，就得交由用户自行承担。

3.扭改用户习惯的需求，一律不考虑

用户行为引导应该是个缓慢而循序渐进的过程。在做技术架构的时候可以稍微激进一点，采用一些新架构新技术去尝试，能提高系统性能；但是在做产品架构的时候，不可冒进。

举例说明：

原有用户账号体系中，不支持手机号注册、登录。在添加这项新功能后，应该是引导用户绑定手机，允许原有方式登录，并增加新的方式登录。尊重原有用户使用用户名的习惯，逐步培养绑定手机号的安全行为，但不能强制用户将登录习惯也改为用手机号进行登录。

因为假设我的用户名为 `wzl` 或者 `willin` 都会比手机号（11位）输入更方便，所以这样的引导并不能帮助用户得到什么益处。不可取。

什么样的需求该重视

1.从运维系统中根据数据结果分析得出的结论

完善运维系统，采集更多需要的信息。根据信息分析得出的可靠结论，才是最重要的需求点。

这里就不举例展开了，一方面数据都是比较私密的，另外一方面，数据所展示出来的问题都是比较明显的。

2.重视有洞见者的每一句话

什么样的人输出什么样的创意。没有偏见，客观陈述。狗嘴里吐不出象牙，所以不要指望肤浅的人给出多么好的意见。而能给出好创意的人，能够源源不断地输出好的创意。

设计，主要来自于思想和经验。

思想这个东西，虽然有后天弥补的空间，但基本都是与生俱来的，可视为先天优势。而经验，则需要知识和实践相结合，可视为后天富足。只有两者都满足，才能成为一个好的设计者。很苛刻，但这是事实。

产品设计

核心思想原则

安全 > 并发性能 > 用户体验(UE) > 用户界面(UI)

这里强调一下并发性能，重于用户体验。原因很简单，因为并发性能直接导致了对服务器硬件环境的要求，所以可视为并发性能即归于成本。没有项目、产品可以不计成本去完善用户体验。

最简化可实行产品（MVP）原则

专注一个突破点。不盲目搞大。

冰冻三尺非一日之寒，一口吃不成胖子。所有庞大的系统，都是由一个个小的子系统逐步演化而来。

明确受众用户，明确核心功能，快速迭代。

系统架构设计

核心思想原则

分治法

即分而治之。

将庞大的计算、存储压力向下级分摊。又可以看做是去中心化的一种实践方式。

数据中心只承担一些核心数据的存储工作；每个服务器都可以存储部分非通用的数据，承担部分的计算及负载压力。下级路由、智能终端设备、智能移动设备等，都可以分摊服务器的压力。

高内聚，低耦合

耦合性与内聚性是模块独立性的两个定性标准，将软件系统划分模块时，尽量做到高内聚低耦合，提高模块的独立性，为设计高质量的软件结构奠定基础。

对外低耦合，对内高内聚

有个例子很容易明白：

> 一个程序有50个函数，这个程序执行得非常好；然而一旦你修改其中一个函数，其他49个函数都需要做修改，这就是高耦合的后果。一旦你理解了它，你编写概要设计的时候设计类或者模块自然会考虑到“高内聚，低耦合”。

1. 耦合、内聚的评估标准是强度，耦合越弱越好，内聚越强越好；
2. 所谓过度指的是由于错误理解导致的效果相反的设计；
3. 耦合指的模块之间的关系，最弱的耦合设计是通过一个主控模块来协调n个模块之间的运作。还是举一个我举过的例子：客户要求在界面上增加一个字段，你的项目要修改几个地方呢？如果你只要修改项目文档，那么你的开发构架就是最低强度的耦合，而这种设计 成熟的开发团队都已经做到了，他们使用开发工具通过项目模型驱动数据库和各层次的代码，而不是直接修改那些代码；
4. 内聚指的是模块内部的功能，最强的内聚就是功能单一到不能拆分，也就是原子化；

5. 所以强内聚和弱耦合是相辅相成的，一个良好的设计是由若干个强内聚模块以弱耦合的方式组装起来的。

前后端分离

参考资料：

- https://segmentfault.com/a/1190000002978095
- http://2014.jsconf.cn/slides/herman-taobaoweb/index.html

注意点： 前后端分离不单指Web的前后端，也包括客户端（前）和服务器（后）的分离。

系统架构设计实践

一个开放平台的设计大概思路。

第一步：定位用户

开发者，分企业开发者和个人开发者。

第二步：系统功能设计

心中先有个梗概，列出列表。

最核心的功能模块：

1. 提供开放接口
2. 提供开放文档
    i. 提供API文档
    ii. 提供SDK
        i. SDK下载，来源各个组，如嵌入式、移动开发、服务器端，提供各种语言的SDK版本
        ii. 除了SDK下载还需要提供SDK使用说明，整合进文档中

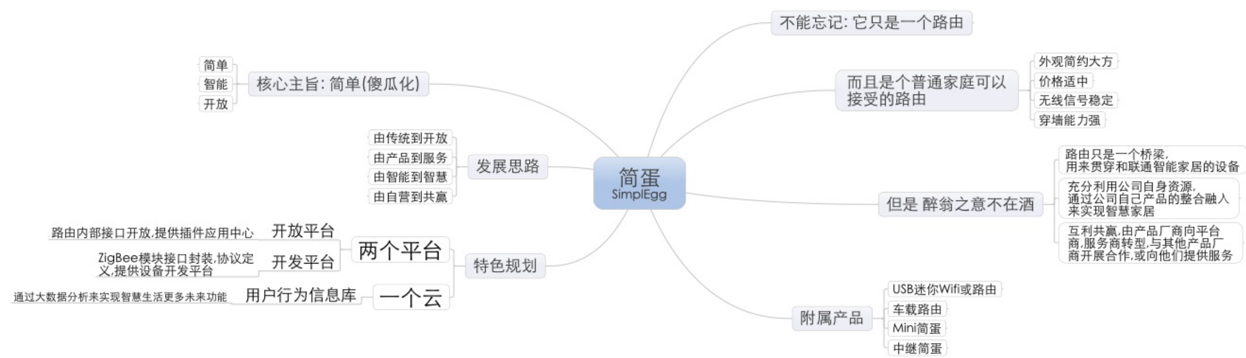
其他功能模块：

1. 用户中心
    i. 开发者认证
2. 产品管理

然后可以搭配脑图、流程图、时序图、用例图等建模工具，设计核心业务模块的流程。

示例

思维导图（脑图）：



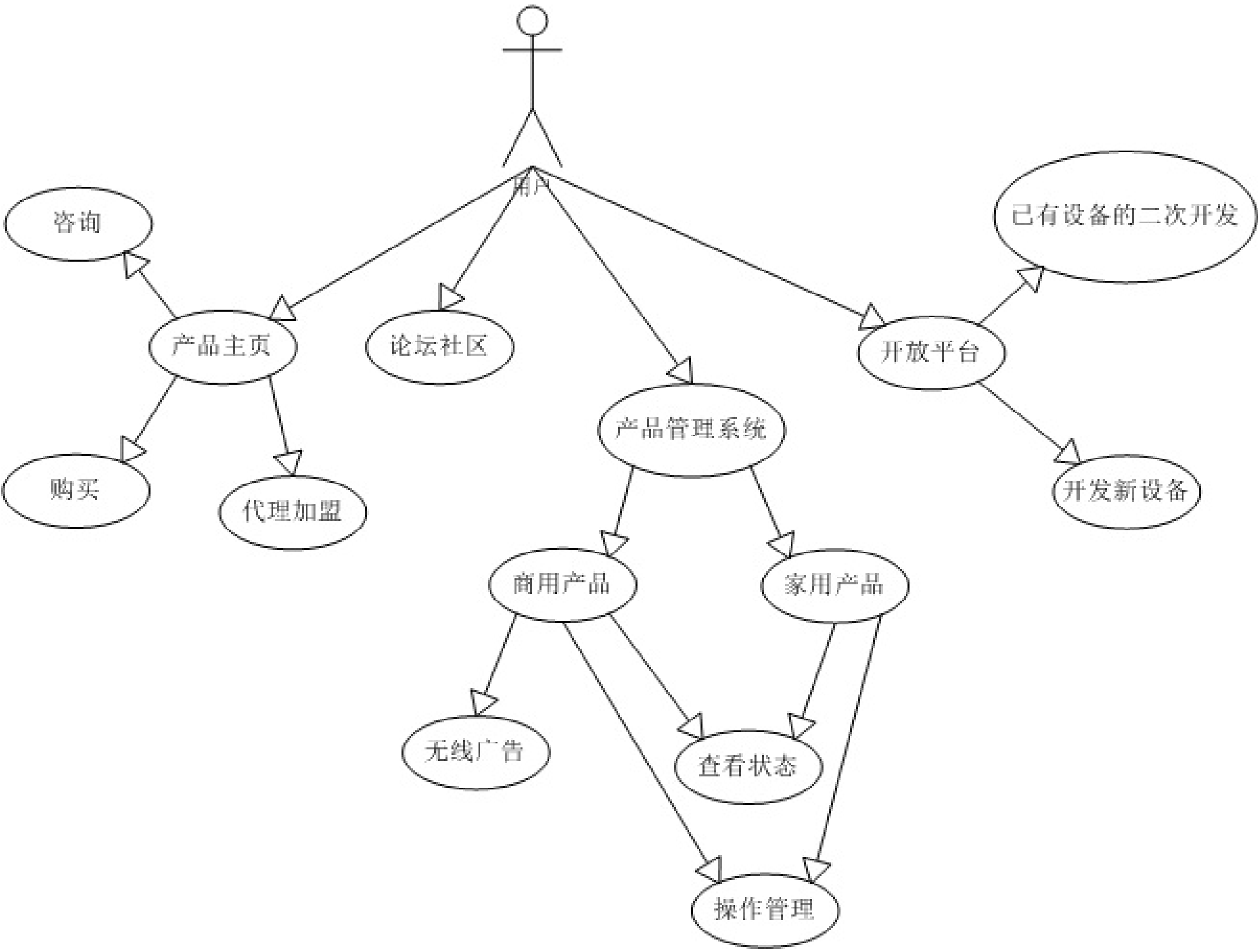
简蛋
SimplEgg
核心主旨: 简单(傻瓜化)
简单
智能
开放
发展思路
由传统到开放
由产品到服务
由智能到智慧
由自营到共赢
特色规划
两个平台
开放平台
路由内部接口开放,提供插件应用中心
开发平台
ZigBee模块接口封装,协议定义,提供设备开发平台
一个云
用户行为信息库
通过大数据分析来实现智慧生活更多未来功能
不能忘记: 它只是一个路由
而且是个普通家庭可以接受的路由
外观简约大方
价格适中
无线信号稳定
穿墙能力强
但是 醉翁之意不在酒
路由只是一个桥梁,用来贯穿和联通智能家居的设备
充分利用公司自身资源,通过公司自己产品的整合融入来实现智慧家居
互利共赢,由产品厂商向平台商,服务商转型,与其他产品厂商开展合作,或向他们提供服务
附属产品
USB迷你Wifi或路由
车载路由
Mini简蛋
中继简蛋

系统结构图：

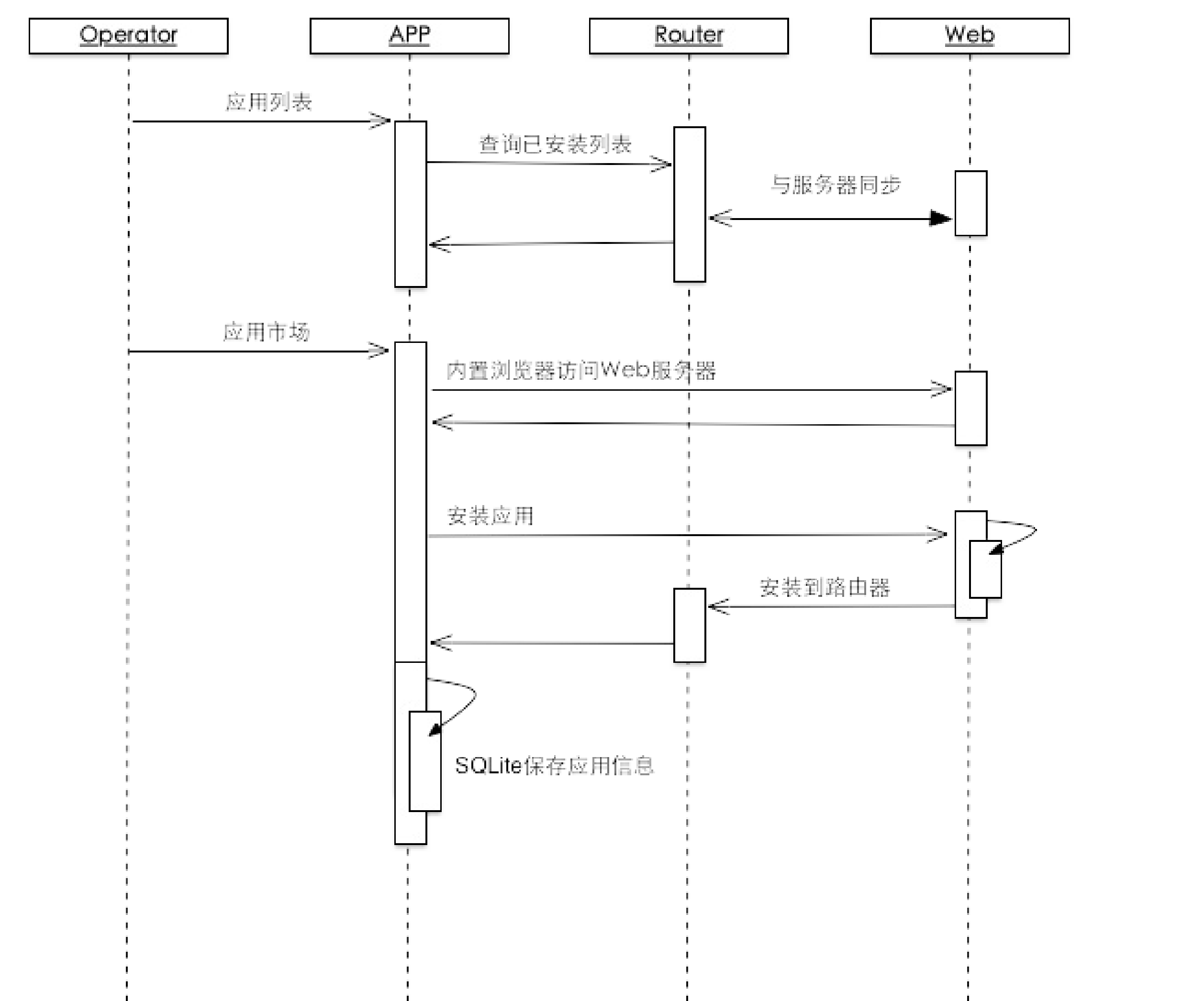


插件开放平台
open.simplegg.com
应用管理
开发者认证
开发证书管理
应用证书校验
插件市场
store.simplegg.com
在线支付
服务器接口
api.jia.to
OAuth 2.0
设备API(Sips)
插件 API
结构化存储服务
路由器接口
client.api.jia.to
(基本与服务器相关接口一致,只能操作单台设备)

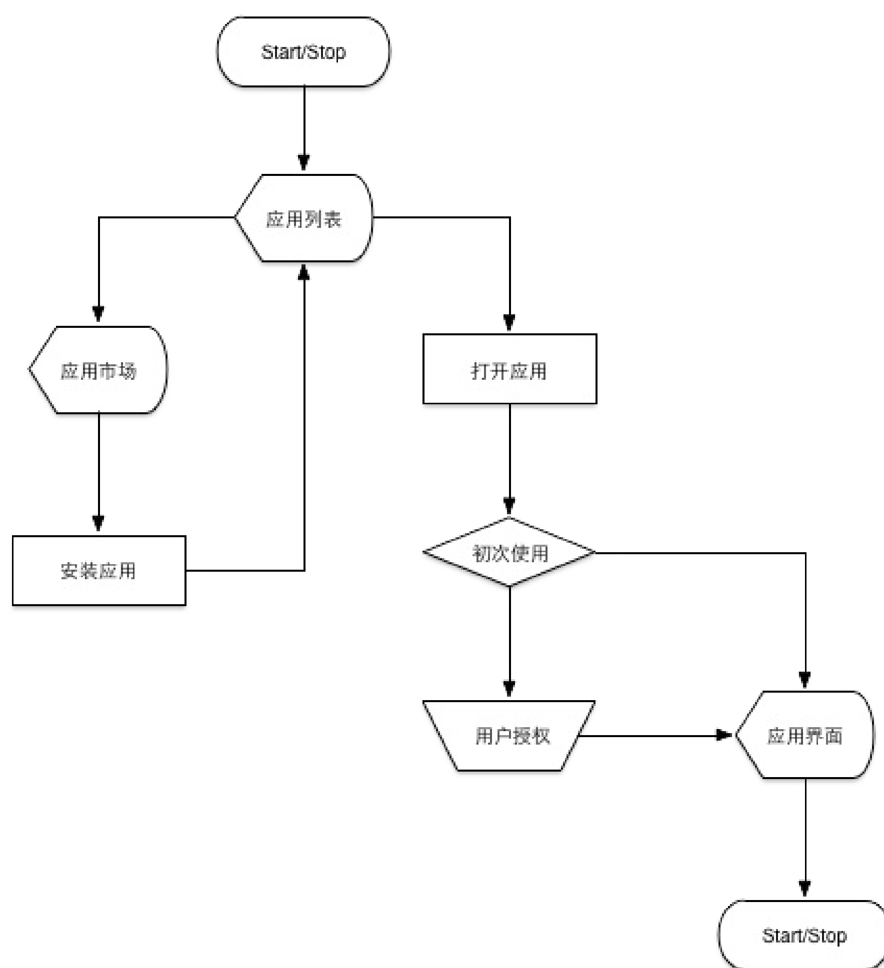
用例图：


用户
咨询
产品主页
论坛社区
产品管理系统
开放平台
已有设备的二次开发
开发新设备
购买
代理加盟
商用产品
家用产品
无线广告
查看状态
操作管理

时序图：


Operator
APP
Router
Web
应用列表
查询已安装列表
与服务器同步
应用市场
内置浏览器访问Web服务器
安装应用
安装到路由器
SQLite保存应用信息

流程图：



第三步：设计数据库表结构

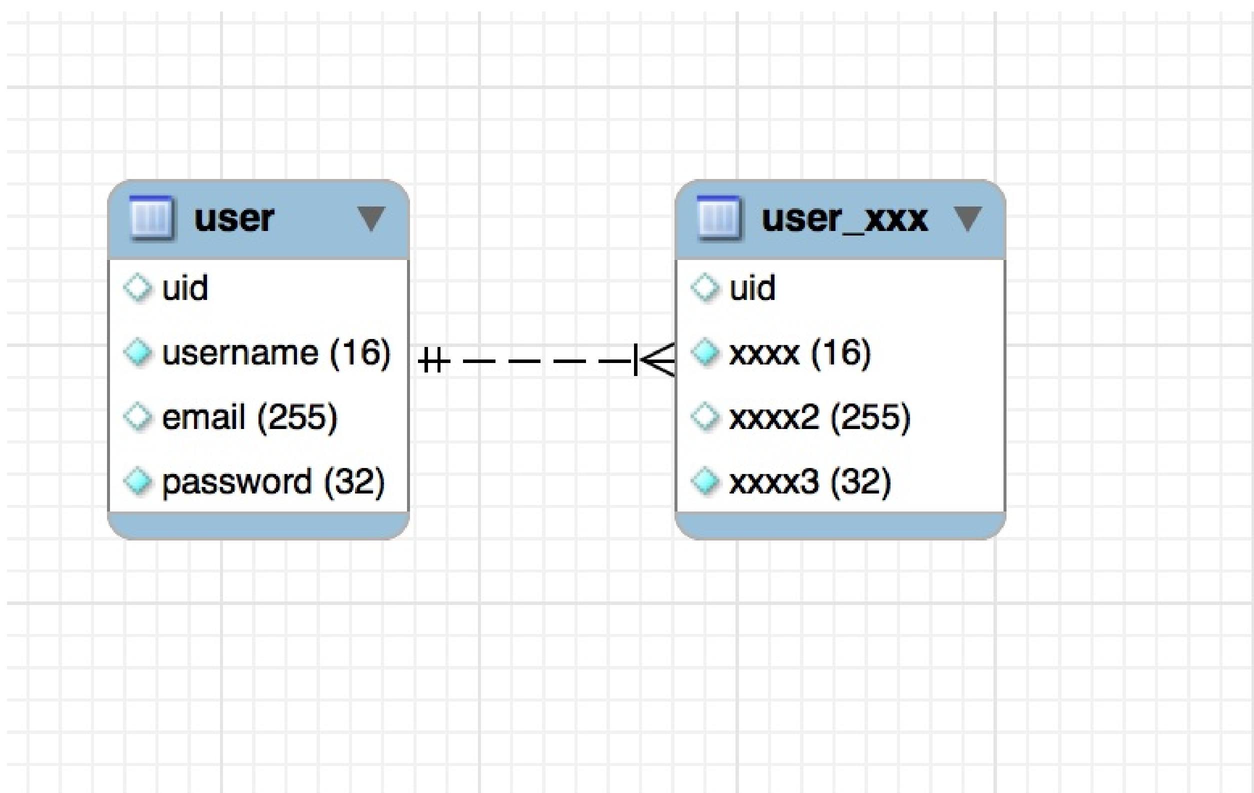
建库建表非常关键。主要原则为，减少冗余数据、避免表字段过多、提高查询性能。

最好是以数字id为主键，避免使用自增id（影响数据同步），外键关系不用外键，关键字段设置索引。

首先第一张表，应该是用户表，虽然不是核心业务，但所有核心业务都与用户相关联，也需要注册登录才能进行。

所以先设计用户表。用户表应该至少两张表，一张是用户基本信息表，只存用户名、密码等或最常用字段，如登录信息；另一张是认证信息，当然也可以分别为企业开发者用户、个人开发者用户建两张表，因为不同的认证方式需要的字段也是不同的。通过用户id字段将用户其他信息表数据进行关联。

示例：



上图为ER图的一个示例，Windows下有 PowerDesigner、Visio等工具，Mac下用MySQLWorkbench。

（可以参考已有系统的用户体系设计，但开发者平台与用户产品系统存在一些细节差异。）

另外，前期也可以考虑加入一些日志表，如认证记录表，存一些历史的认证信息。根据项目时间预算，如果前期不考虑，后期也需要考虑加上。

第四步：搭建系统框架

先搭建一个大框架，配置缓存数据库，加入通用类，配置端口，并且能够运行。

（可以参考现有的项目以及项目章节的结构）

搭建测试框架（如果项目进度预算允许）。在项目实现过程中的细节，关注下一章节的内容中讲解。

第五步：迭代

重复上面的过程，完善新功能模块设计，加入到已有系统。

项目

人无远虑必有近忧。能在设计过程中解决掉的问题，绝对不要拖到实现过程中迭代。

推荐配置

参考： https://github.com/w2fs/best-practice

现有项目

内部信息，登录Git访问： http://172.18.2.179/doc/project/tree/master

项目结构

目录分配尽可能简洁、清晰。

服务器端：MVC模式

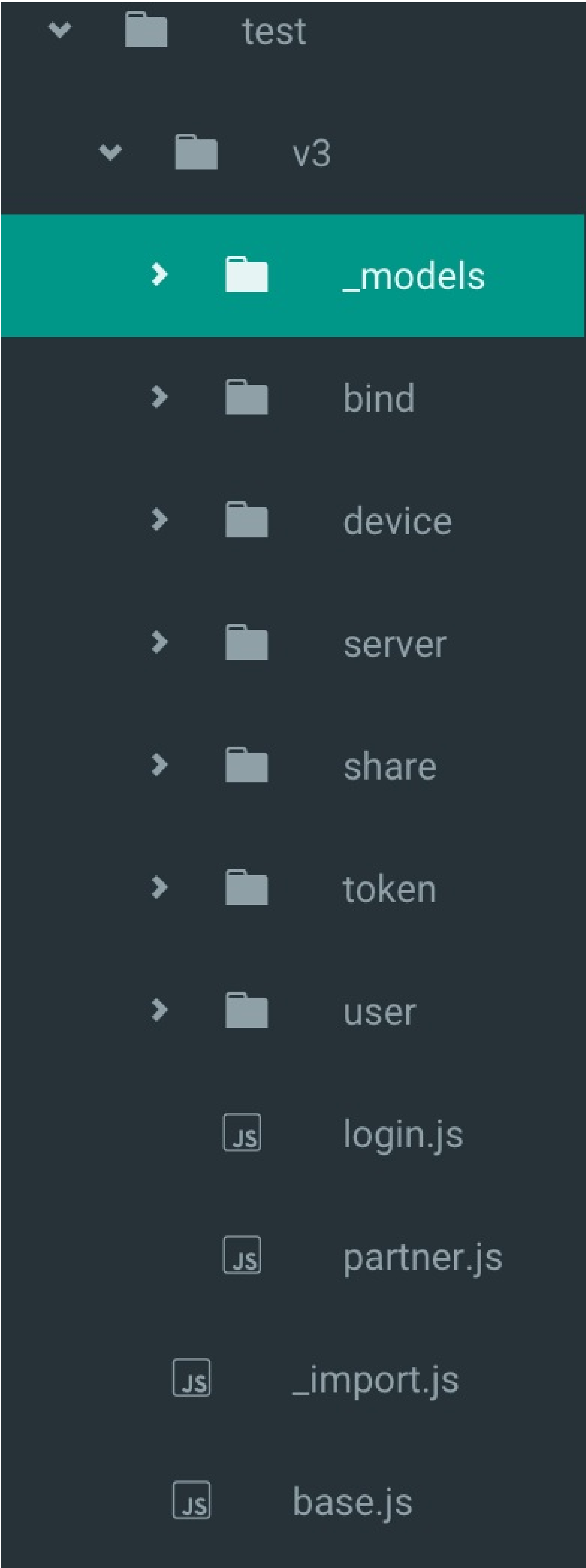
项目目录结构：

```
.
├── bin
│   └── # 可执行文件
├── config
│   └── # 配置文件
├── lib
│   └── # 通用类
├── locales
│   └── # 语言文件
├── package.json
├── routes
│   └── v3 (子项目)
│       ├── handlers
│       │   └── # Controller
│       ├── models
│       │   └── # Model
│       └── views (接口可无，另推荐前后端分离)
│           └── # View
└── test
```

客户端：MVVM模式

```
.
├── app
│   ├── app.js
│   └── index.html
├── main.js
└── src
    ├── app.js
    ├── components
    │   └── # View Model
    ├── index.less
    ├── model
    │   └── # Model
    ├── routes
    │   └── # View
    ├── webpack.config.js
    └── webpack.config.prod.js
```

测试：BDD


test
v3
_models
bind
device
server
share
token
user
login.js
partner.js
_import.js
base.js

上图为BDD测试目录，非测试用例文件（或目录）以 `_` 开头。

HAPI

Server

```
import hapi from 'hapi';

// Static File Server
import inert from 'inert';
// Render Views
import vision from 'vision';

const server = new hapi.Server();

server.connection({
  host: '127.0.0.1',
  port: 4000,
  router: {
    stripTrailingSlash: true
  }
}, { timeout: { server: 5000, socket: 5000 } });

// 根据需要注册插件
server.register([inert, vision], () => {
  server.start(() => {
    console.log(`Server started at:  ${server.info.uri}`);
  });
});

// Load Routes
server.route(require('./routes'));

// Error Response Handler
server.ext('onPreResponse', (request, reply) => {
  const response = request.response;
  if (!response.isBoom) {
    return reply.continue();
  }
```

```
  // return custom err result
});

// Add Templates Support with handlebars
server.views({
  path: `${__dirname}/lib/views`,
  engines: { html: require('handlebars') },
  partialsPath: `${__dirname}/lib/views/partials`,
  isCached: false
});

module.exports = server;
```

Plugins

- 自动文档：https://github.com/WulianCC/hapi-swagger
- 用户鉴权：http://hapijs.com/api#serverauthapi
- 表单校验：https://github.com/hapijs/joi
- HTTP错误：https://github.com/hapijs/boom
- 渲染模板页面：https://github.com/hapijs/vision
- 打印错误：https://github.com/hapijs/good
- 静态文件引用：https://github.com/hapijs/inert

其中，用户鉴权可参考：https://code.aliyun.com/shgg/push/blob/master/lib/auth.js

Socket.IO

http://socket.io/

Server

应用场景：服务器间通信。

```
const io = require('socket.io')().listen(6666);

io.on('connection', (socket) => {
  socket.on('client', async(data) => {
    // Codes Here
  });
});

exports.io = io;
```

Client

```
import io from 'socket.io-client';

const socket = io('ws://127.0.0.1:6666/');

// 上线汇报
socket.emit('client', ()=>{
  return 'Hello World'
});
socket.on('server', async(data) => {
  // 处理服务器消息
});
```

Later

http://bunkat.github.io/later/index.html

应用场景：计划任务，类似Crontab。

```
import later from 'later';
// 每分钟执行一次
later.setInterval(()=>{
  // Codes

}), later.parse.recur().every(1).minute());
// 每天的 16:55 执行
later.setInterval(()=>{
  // Codes

}), later.parse.cron('55 16 */1 * * ?'));
// 每小时的 1 分 执行
later.setInterval(()=>{
  // Codes

}), later.parse.cron('1 */1 * * * ?'));
```

Electron

目前支持：Mac、Win、Linux三个平台。

快速示例： https://github.com/electron/electron-quick-start

打包工具

electron-packager: https://github.com/electron-userland/electron-packager

Client

目录结构

```
.
├── app
│   ├── app.js
│   └── index.html
├── main.js
├── package.json
└── src
    ├── app.js
    ├── components
    │   ├── delete.js
    │   ├── login.js
    │   └── main.js
    ├── index.less
    ├── routes
    │   └── index.js
    ├── webpack.config.js
    └── webpack.config.prod.js
```

运行脚本

```
"scripts": {
  "start": "./node_modules/.bin/webpack --config src/webpack.confi
  "test": "./node_modules/.bin/webpack --config src/webpack.config.
  "pack-win": "./node_modules/.bin/electron-packager . --asar  --ov
  "pack-mac": "./node_modules/.bin/electron-packager . --asar --ove
  "pack-all": "./node_modules/.bin/electron-packager . --out=dist -
}
```

UDP

官方文档： https://nodejs.org/api/dgram.html

中文翻译： http://shouce.w3cfuns.com/nodejs/dgram.html

Server

应用场景： 心跳服务。

```
import dgram from 'dgram';

const server = dgram.createSocket('udp4');
server.on('error', (err) => {
  // 处理错误
  server.close();
  server.bind(6666);
});

server.on('message', async(msg, info) => {
  // 处理消息
});

server.on('listening', () => {
  const address = server.address();
  console.log('Push Client Server listening at %s - %s', `${address
});
server.bind(6666);
```

ES 6/7

Async

```
async function fn(args){
  // ...
}

// 等同于

function fn(args){
  return spawn(function*() {
    // ...
  });
}
```

多个 `await` 命令后面的异步操作，如果不存在继发关系，最好让它们同时触发。

```
let foo = await getFoo();
let bar = await getBar();
```

上面代码中， `getFoo` 和 `getBar` 是两个独立的异步操作（即互不依赖），被写成继发关系。这样比较耗时，因为只有 `getFoo` 完成以后，才会执行 `getBar` ，完全可以让它们同时触发。

```
// 写法一
let [foo, bar] = await Promise.all([getFoo(), getBar()]);

// 写法二
let fooPromise = getFoo();
let barPromise = getBar();
let foo = await fooPromise;
let bar = await barPromise;
```

上面两种写法，`getFoo` 和 `getBar` 都是同时触发，这样就会缩短程序的执行时间。

Proxy

```
node --harmony-proxies
```

示例代码：

```
const proxy = new Proxy({}, {
  get: (target, property) =>
    (test) => [target, property, test]
});

console.log(proxy.func);          // [Function]
console.log(proxy.func('123'));   // [ {}, 'func', '123' ]
```

Benchmark

性能对比测试框架 Matcha： https://github.com/logicalparadox/matcha

使用场景

技术选型，如图形验证码，在NPM包选取使用 `canvas` 还是 `ccap` 时可以用。

或，一个问题，有多种解决方案，选择采用哪一种方案的时候。

注意：所有需要做选择的场景，最好都先做一下对比。

结果报告示例

```
ATL (After v1.0.1)
  if > (true) .................................. 4,752,967 op/s
  if = (true) .................................. 4,653,896 op/s
  if < (false) ................................. 4,612,560 op/s

Left Shift (ATL v1.0.0)
  << > (true) .................................. 2,562,098 op/s
  << = (true) .................................. 2,473,787 op/s
  << < (false) ................................. 2,458,286 op/s
```

示例代码

```
suite('ATL', function() {
  bench('if > (true)', function(){
    atl('1.6.7','1.4.4');
  });
  bench('if = (true)', function(){
    atl('1.4.4','1.4.4');
  });
  bench('if < (false)', function(){
    atl('1.1.6','1.4.4');
  });
});

suite('Left Shift', function(){
  bench('<< > (true)', function(){
    atls('1.6.7','1.4.4');
  });
  bench('<< = (true)', function(){
    atls('1.4.4','1.4.4');
  });
  bench('<< < (false)', function(){
    atls('1.1.6','1.4.4');
  });
});
```

源码位于: https://github.com/WulianCC/node-atl/blob/master/benchmark/parse.js

Test

谁开发，谁测试。

注意： 原则上应该先写测试，再进行编码；如果需求时间紧，可以先进行功能实现，但务必后续维护时候将测试代码补充完善。

BDD（优先）+TDD（完全代码覆盖）

测试框架

- v2: mocha + istanbul
- v3: ava + nyc

下文示例来自v2中的测试，源码位于:
https://code.aliyun.com/shgg/v2/tree/c605998dc59d1b3e1d91681cc9b0c9daec4ef341/test

(内部代码，须登录Git)

TDD

Test Driven Development，（单元）测试驱动开发。

特点：

1. 直接引用对应源码，执行方法进行测试；
2. 测试用例须设计完整，把所有分支都Cover到。

示例：

```
describe('Lib Common', function () {
    'use strict';
    it('isEmpty', function () {
        // isObject
        isEmpty({}).should.be.equal(true);
        isEmpty([]).should.be.equal(true);
        isEmpty({a: 1}).should.be.equal(false);
        isEmpty([1, 2]).should.be.equal(false);
        // isString
        isEmpty('').should.be.equal(true);
        isEmpty('sth').should.be.equal(false);
        // isNumber
        isEmpty(0).should.be.equal(true);
        isEmpty(0.1).should.be.equal(false);
        // null and undefined
        isEmpty(null).should.be.equal(true);
        isEmpty(undefined).should.be.equal(true);
        // boolean
        isEmpty(false).should.be.equal(true);
        isEmpty(true).should.be.equal(false);
        // 最后一行false
        isEmpty(isEmpty).should.be.equal(false);
    });
    it('md5/sha1', function () {
        md5('sth').should.equal('7c8db9682ee40fd2f3e5d9e71034b717'
        sha1('sth').should.equal('dec981e3bbb165d021029c42291faf06
    });
    it('authcode', function () {
        authcode(authcode('test'), 'DECODE').should.be.equal('test
        authcode(authcode('test', 'ENCODE', 'key'), 'DECODE', 'key
        authcode('c008AsZqmGL8VuEVpZKVlbPwXzSsCZ+YX5K5CAGpMMqn', '
    });
});
```

BDD

Behavior Driven Development，行为驱动开发。

特点：

1. 运行系统，模拟用户请求进行访问；
2. 行为分析要完整，要将可能所有结果覆盖。

示例：

```
/* 测试路由 */
app.get('/test/model/mysql/init/ok', function (req, res) {
    'use strict';
    return db.opensips('v1/subscriber').then(
        function () {
            res.send(200, 'ok');
        }).catch(function (err) {
            logger('routes/test/model/mysql/ok', err);
            res.send(403, 'fail');
        });
});

app.get('/test/model/mysql/init/fail', function (req, res) {
    'use strict';
    return db.opensips('test/notExisted').then(
        function () {
            res.send(200, 'OK');
        }).catch(function () {
            res.send(200, 'fail');
        });
});


/* 测试脚本 */
describe('Demo', function () {
    'use strict';
    it('404 not found', function (next) {
        request(app)
            .get('/sth/not/exist')
            .set('Accept', 'text/plain')
            .expect(200)
            .end(function (err, res) {
                if (err) {
                    throw err;
```

```
                }
                should(res.body.status).be.equal(0);
                next();
            });
    });
    it('403 not allowed', function (next) {
        request(app)
            .get('/v2/basic/mqtt')
            .set('Accept', 'text/plain')
            .expect(200)
            .end(function (err, res) {
                if (err) {
                    throw err;
                }
                should(res.body.status).be.equal(0);
                next();
            });
    });
    it('Init opensips/subscriber Should be OK', function (next) {
        request(app)
            .get('/test/model/mysql/init/ok')
            .set('Accept', 'text/plain')
            .expect(200)
            .expect('ok')
            .end(function (err) {
                if (err) {
                    //console.log(res.body);
                    throw err;
                }
                next();
            });
    });
    it('Init test/subscriber Should be FAILED', function (next) {
        request(app)
            .get('/test/model/mysql/init/fail')
            .set('Accept', 'text/plain')
            .expect(200)
            .expect('fail')
            .end(function (err) {
                if (err) {
```

```
                    //console.log(res.body);
                    throw err;
                }
                next();
            });
    });
});
```

ES6 下的 BDD 测试示例对比：

```
import {test, server, assert} from './_import';
let location;
test.before(async() => {
  const response = await server.inject({
    method: 'POST',
    url: '/login',
    payload: {
      username: 'willin',
      password: 'PASSWORD'
    }
  });
  location = response.headers.location;
});

test('GET / 302', async() => {
  const response = await server.inject({
    method: 'GET',
    url: '/'
  });
  assert.equal(response.statusCode, 302);
});

test('GET /login 200', async() => {
  const response = await server.inject({
    method: 'GET',
    url: '/login'
  });
  assert.equal(response.statusCode, 200);
});
```

```
test('POST /login 302', async() => {
  const response = await server.inject({
    method: 'POST',
    url: '/login',
    payload: {
      username: 'willin',
      password: 'PASSWORD'
    }
  });
  assert.equal(response.statusCode, 302);
});

test('POST /login 401', async() => {
  const response = await server.inject({
    method: 'POST',
    url: '/login',
    payload: {
      username: 'willin',
      password: 'Ww10842073305zZa28v3PO5Ok0L63IdA'
    }
  });
  assert.equal(response.statusCode, 401);
});

test('POST /login Invalid Params 403', async() => {
  const response = await server.inject({
    method: 'POST',
    url: '/login',
    payload: {
      username: 'willin'
    }
  });
  assert.equal(response.statusCode, 403);
});

test('GET /doc 200', async() => {
  const response = await server.inject({
    method: 'GET',
```

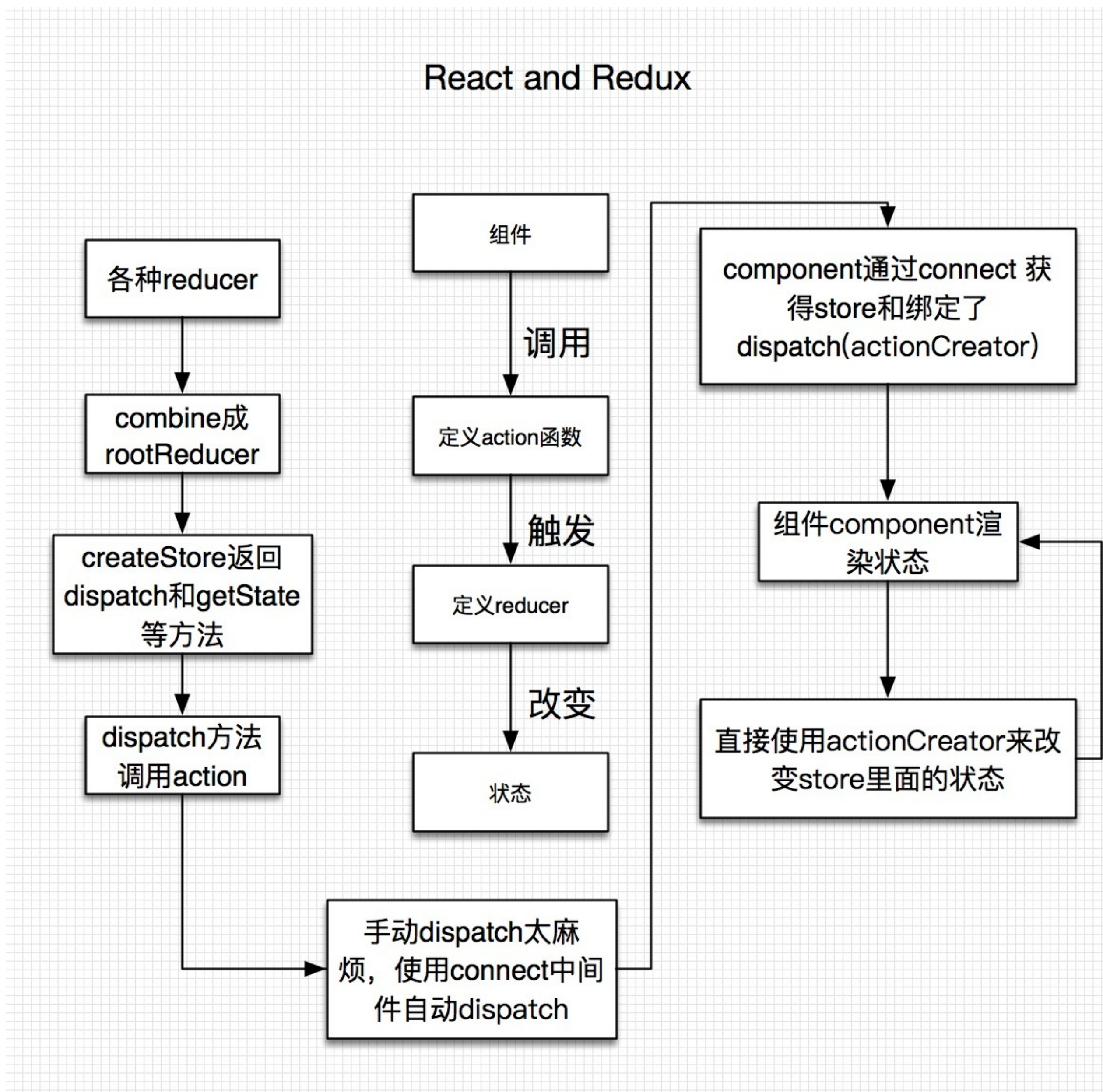
```
    url: location
  });
  assert.equal(response.statusCode, 200);
});

test('GET /doc 302', async() => {
  const response = await server.inject({
    method: 'GET',
    url: '/doc?'
  });
  assert.equal(response.statusCode, 302);
});
```

React and Redux

> view层发出actions通知触发store里的reducer从而来更新state；state的改变会将更新反馈给我们的view层，从而让我们的view层发生相应的反应给用户。

流程图



目录结构

目录结构大概可以这样规划

```
app
  |_components
  |_reducers
  |_actions
  |_stores
    |_configureStores.js
  |_main.js
```

核心代码

举例核心代码。值得注意的是其中有一个state的传递有一些迷惑的地方，在下面的注释中可以找到思路。

components

```
import React, { Component } from 'react';
import { bindActionCreators } from 'redux';
import { connect } from 'react-redux';
import * as CounterActions from '../../actions/counter.js';

class Counter extends Component {
  constructor() {
    super();
    this.state = {};
  }
  componentWillMount = () => {
    this.startCount();
  }
  startCount = () => {
    const { actions } = this.props;
    actions.listen('INC');
  }
  render() {
    return (
      <div>
```

```
        {this.props.counter.count}
      </div>
    );
  }
}
Counter.propTypes = {
  actions: React.PropTypes.object.isRequired,
  counter: React.PropTypes.object.isRequired
};
// 声明 connect 连接
// 将 redux 中的 state传给 App
function mapStateToProps(state) {
  return {
    counter: state.counter
  };
}
function mapDispatchToProps(dispatch) {
  const boundCounter = bindActionCreators(CounterActions, dispatch)
  return {
    actions: Object.assign({}, boundCounter)
  };
}
// 声明 connect 连接
export default connect(mapStateToProps, mapDispatchToProps)(Counter
```

actions

action函数必须返回一个带有type属性的plain object。

```
import * as constants from '../../constants/counter';
export function listen(type) {
  switch (type) {
    case 'INC':
      return {
        type: constants.INC
      };
    case 'DEC':
      return {
        type: constants.DEC
      };
  }
}
```

reducers

reducer就是迎接action函数返回的线索的数据再处理函数，action是预处理函数。

```
import {INC, DEC} from '../../constants/counter';
// 初始状态
const initState = {
  count: 1
};
// 定义转换的reducer函数
// action = {type: 'INC',counter: {count: 1}};
export default function start(state = initState, action) {
  switch (action.type) {
    case INC:
    // 对这个action做出响应
      state.count += 1 ; // 改变状态
      // return {count: 2} 返回给页面;
      return state;
    case DEC:
    // 对这个action做出响应
      state.count -= 1; // 改变状态
      // return {count: 0} 返回给页面;
      return state
    default:
      return state;
  }
}
```

```
import { combineReducers } from 'redux';
import counter from './counter'; // {count: 2}返回给页面，所以页面用的
// 通过combineReducers将多个reducer合并成一个rootReducer:
const rootReducer = combineReducers({
  counter, // {count: 1}
  others
});
export default rootReducer;
```

MySQL

查询优化

LIMIT 1

单条数据查询

```
SELECT uid FROM ?? WHERE email = ? LIMIT 1
```

或单条记录更改

```
UPDATE ?? SET lastonline = if(updatedat < ? , ? , lastonline), ? WH
```

或单条记录删除，加 `LIMIT 1` 。

SELECT 嵌套 SELECT

如：

```
SELECT did,type,
  (select username from ?? as t1 where t1.uid = fromuid LIMIT 1) us
  (select email from  ?? as t1 where t1.uid = fromuid LIMIT 1) emai
FROM ?? WHERE --xxx
```

优化为：

```
SELECT t1.`type`,t1.did,t2.username,t2.email FROM ??
  LEFT JOIN ?? ON t1.touid=t2.uid
WHERE --xxx
```

多次JOIN

```
SELECT
    `t1`.`xxx`,
    `t1`.`xxx`,
    `t2`.`xxx`,
    `t2`.`xxx`,
    `t3`.`xxxx`,
    `t3`.`xxx`,
    `t4`.`xxx`,
    `t5`.`xxxx`
FROM (((?? `t1`
  left join ?? `t2` on((`t1`.`did` = `t2`.`did`)))
  left join ?? `t3` on((`t1`.`did` = `t3`.`did`)))
  left join ?? `t4` on((`t4`.`username` = `t1`.`did`)))
  left join ?? `t5` on((`t1`.`did` = `t5`.`did`)))
WHERE --xxx;
```

80条记录结果的查询约40s，拆分查询，t1-t3主要查询，t4、t5表的数据只在部分记录中需要，分别做两次查询，共计三次查询，优化后查询总耗时1s以内。

表结构优化

引擎

如果需要用事务用 `InnoDB` 。

如果对查询效率要求高用 `MyISAM` 。

表结构优化

基于 `MyISAM` 引擎。

- 避免使用自增ID；
- 避免使用 `varchar` ，而用 `char` ；
- 避免使用 `text` ，而用 `blob` ；
- 避免使用 `外键` ；

- 不允许空 `null` ；
- 如果查询的WHERE条件有多个字段，应该创建 `联合索引` 。

其他

阿里云RDS DMS工具： https://dms-rds.aliyun.com/?host=

阿里云RDS性能优化工具：

管理控制台 产品与服务

产品与服务
云服务器 ECS
云数据库 RDS 版
对象存储 OSS
CDN
访问控制
云监控
用户中心
账号管理
费用中心
续费管理
消息中心
备案管理

基本信息
帐号管理
数据库管理
数据库连接
监控与报警
数据安全性
服务可用性
日志管理
性能优化
备份恢复
参数设置

杭州
首届阿里在线技
性能优化
慢日志统计
选择时间范围：
时间
2016-08-16
2016-08-16
2016-08-16

Redis

代码示例

一般Redis里存储的数据需要一个默认的TTL，即到期删除，尽可能避免无用数据长期存储。

```
import redis from 'wulian-redis';

const client = redis({
  host: '127.0.0.1',
  port: 6379,
  db: 0
});

(async()=>{
  // 推荐
  await client.set('trial:127.0.0.1', 1, 900);

  // 或

  // 需要注意，如果该`key`之前已存在，且ttl已设置，重新set之后，ttl会变成-1
  await client.set('trial:127.0.0.1', 1);
  // TTL: 900s
  await client.expire('trial:127.0.0.1', 900);
})();
```

注意事项

- 设置TTL，默认超时时间
- Value值为字符串，如果JSON数据存之前要 `JSON.stringify` ，取之后要 `JSON.parse`
- 具体Redis命令参数参考 http://redis.io/commands

常用工具

proxychains-ng

GFW工具： https://github.com/rofl0r/proxychains-ng

Mac下安装命令为：

```
brew install proxychains-ng
```

autojump

专业命令行跳转工具，谁用谁知道： https://github.com/wting/autojump

```
brew install autojump
```

配合 Oh My Zsh `autojump` 插件使用。

HTTP/2 and SPDY indicator

检查是否支持HTTP/2或SPDY的Chrome插件：
https://chrome.google.com/webstore/detail/http2-and-spdy-indicator/mpbpobfflnpcgagjijhmgnchggcjblin

Wappalyzer

检查网页使用了哪些常见技术的Crhome插件：
https://chrome.google.com/webstore/detail/wappalyzer/gppongmhjkpfnbhagpmjfkannfbllamg

Babel

http://babeljs.io/

`babel-node` 和 `babel-register` 功能基本相近。

配置文件参考： https://github.com/w2fs/best-practice

Babel-Register

项目内安装

```
npm install babel-register --save
```

使用：

创建 `babel.js`

```
require('babel-register');
module.exports = require('./server.js');
```

执行：

```
node babel.js
```

Babel-Node

全局安装：

```
npm install babel-cli -g
```

使用：

```
babel-node xxx.js
```

编译 ES5 代码

```
babel src --out-dir dist
```

源目录 `src` ，目标目录 `dist` 。

PM2

https://github.com/Unitech/pm2

当前使用场景：产品环境守护进程。

常用命令

启动项目

```
pm2 start xxx.js    #直接启动入口文件

pm2 start xxx.json #通过配置启动
```

配置文件参考：

```
[{
    "name": "app",
    "script": "babel.js",
    "log_date_format": "YYYY-MM-DD HH:mm:ss Z",
    "cwd": "/home/project",
    "error_file": "/home/project/logs/app.err.log",
    "out_file": "/home/project/logs/app.out.log",
    "max_memory_restart": "800M",
    "instances": 0,
    "exec_mode": "cluster",
    "merge_logs": true,
    "env": {
        "NODE_ENV": "production"
    }
},{
  "script" : "./examples/child.js",
  "error_file" : "errLog.log",
  "out_file" : "outLog.log",
  "pid_file" : "child",
  "instances" : "4",
  "min_uptime" : "10",
  "max_restarts" : "4"
},{
  "script" : "examples/env.js",
  "error_file" : "errEcho.log",
  "out_file" : "outEcho.log",
  "name" : "ok",
  "pid_file" : "echo.pid",
  "max" : "1",
  "exec_mode" : "cluster_mode",
  "port" : "9001",
  "env_variable" : "TOTO",
  "TEST_VARIABLE" : "YESSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSIR"
}]
```

重启项目

```
pm2 restart #PID# #重启一个进程

pm2 restart app-name #重启一个应用

pm2 restart all #重启所有项目
```

停止

```
pm2 stop #PID#

pm2 stop app-name
```

终止

```
pm2 delete app-name #删除一个进程

pm2 kill #终止所有进程
```

开机自启

```
pm2 startup
```

参考官方文档获取详细使用说明。

开发指南

产品研发流程

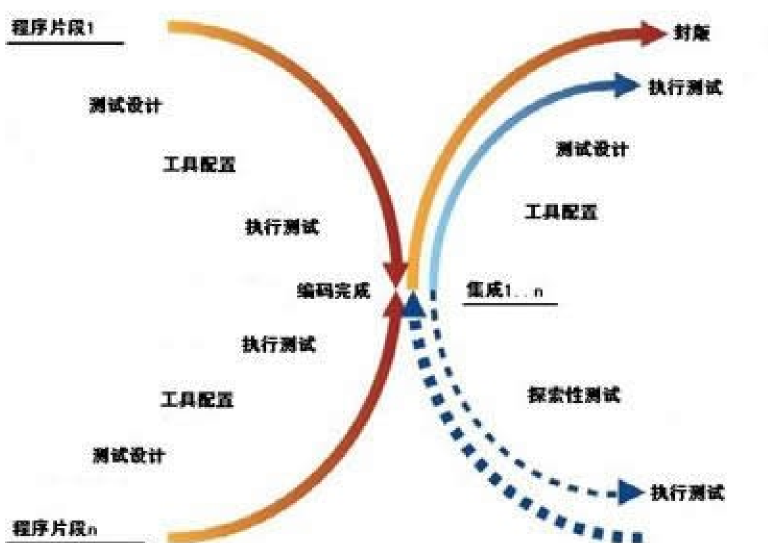
> 需求 -> 设计 -> 开发 -> 测试 -> 验收

开发环节中，需要包含中间的三个过程，设计、开发、测试。

设计 > 测试 > 开发 (重要程度)

不经过思考的代码是站不住脚的。

任何功能模块在编写代码之前，最重要的是理解业务流程，将其用流程图、时序图或其他方式表达出来，参考设计图和相关记录文档文字进行开发。做到先设计，后编码。



基本过程：

1. 明确当前要完成的功能，可以记录成一个 TODO 列表。
2. 快速完成针对此功能的测试用例编写。
3. 测试代码编译不通过。
4. 编写对应的功能代码。
5. 测试通过。
6. 对代码进行重构，并保证测试通过。
7. 循环完成所有功能的开发。

下面的子章节将通过实践讲解，如何搭建一个用户账号体系的完整过程。

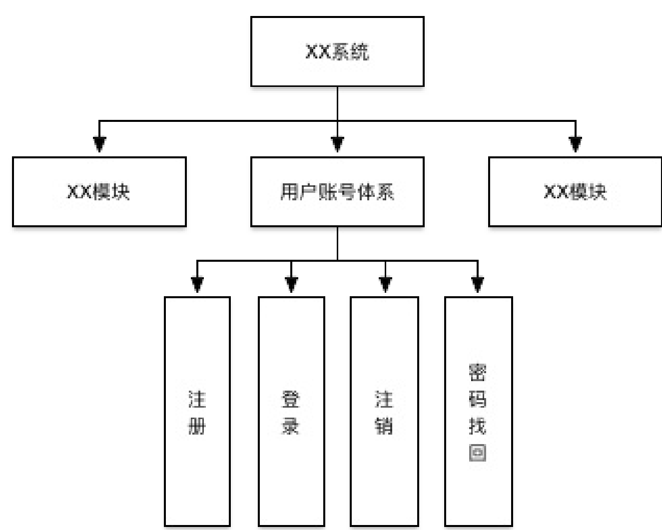
功能模块设计

实现 `注册、登录、注销、密码找回` 的需求。

用户可以通过手机号进行登录和密码找回。

系统结构图

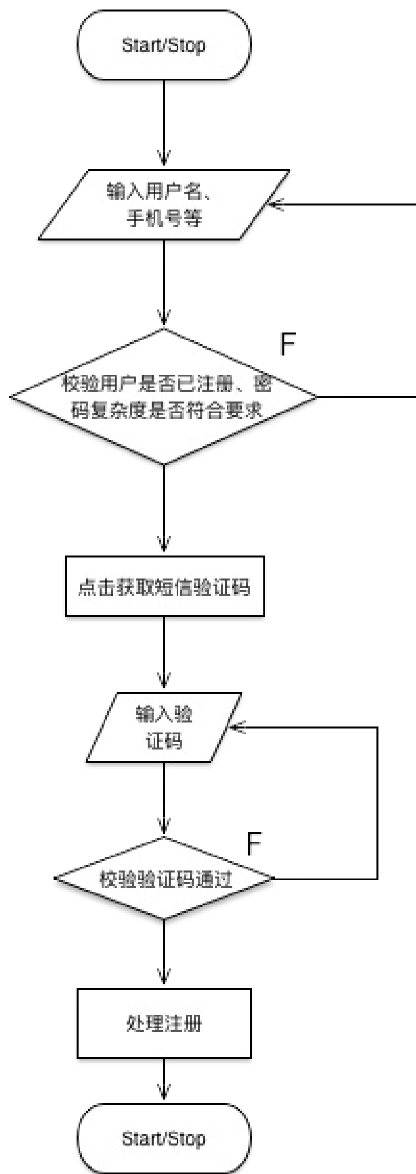
示例：



功能模块

注册

流程图，示例：



涉及参数：

- 用户名
- 密码
- 手机号
- 短信验证码

约束条件：

- 短信验证码发送频率限制 90s
- 注册频率限制每30分钟只能注册 1次

（示例，根据实际需求和业务进行约束）

登录

涉及参数：

- 用户名或手机号
- 密码

约束条件：

- 30分钟内 连续出错 3次 限制登录

找回密码

涉及参数：

- 手机号
- 短信验证码
- 新密码

数据库表结构设计

ER图，略。

用户基本信息表

示例：

```
CREATE TABLE `user` (
  `uid` int(11) unsigned NOT NULL COMMENT '用户ID',
  `username` char(16) NOT NULL DEFAULT '' COMMENT '用户名',
  `password` char(32) NOT NULL DEFAULT '' COMMENT '密码',
  `salt` char(8) NOT NULL DEFAULT '' COMMENT '加盐加密',
  `mobile` char(16) NOT NULL DEFAULT '' COMMENT '手机号',
  `createdat` int(10) unsigned NOT NULL COMMENT '注册时间',
  `updatedat` int(10) unsigned NOT NULL COMMENT '更新时间',
  PRIMARY KEY (`uid`),
  UNIQUE KEY `username` (`username`),
  UNIQUE KEY `mobile` (`mobile`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8;
```

说明：

1. `uid` 主键没有设置自增id，可以随机分配，但需要在不同数据库上分号段注册，以及需要判断是否已注册
2. `password` 密码不能直接 `MD5` 或 `SHA1` 加密存储，需要加盐加密
3. `createdat` 表示注册时间， `updatedat` 表示密码修改、手机绑定等更新时间
4. 用户名和手机号均为唯一字段
5. 养成加注释的习惯

用户附加信息表

将非基本信息，如开发者资料、用户详细资料、认证、等级等存入用户附加信息表（也可建多个附加信息表，如用户认证表、用户配置表等）。

示例：

```
CREATE TABLE `usermeta` (
  `uid` int(11) unsigned NOT NULL COMMENT '用户ID',
  `truename` char(16) NOT NULL DEFAULT '' COMMENT '真实姓名',
  `gender` enum('male','famale') NOT NULL DEFAULT 'male' COMMENT '
  `verified` tinyint(1) unsigned NOT NULL DEFAULT '0' COMMENT '实名
  PRIMARY KEY (`uid`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8;
```

说明：

1. 如果需要按姓名或性别查找，将 `truename` 或 `gender` 字段加索引
2. 查找一个用户是否已经完成实名认证，可以用 `SELECT t2.verified FROM user t1 LEFT JOIN usermeta t2 ON t1.uid=t2.uid WHERE t1.username = ? LIMIT 1` 方式查询

其他表

部分信息不需要存入数据库，如日志，可以存在log文件中；登录信息，可以用Redis等缓存存储。综合考虑性能、成本及服务器配置决定。

再讲解一种，针对 `开发者` 的表设计。

举例，开发者一般有两种类型，企业开发者、个人开发者。

那么，可以在 `usermeta` 表里加一个开发者类型字段， `dev_type` ：

```
`dev_type` enum('personal','enterprise') NOT NULL DEFAULT 'personal
```

另外建两张表，为 `developer_personal` 和 `developer_enterprise` 分别存放个人开发者和企业开发者的相关信息字段。

可以在系统的业务逻辑里加入一些限制，如个人开发者可以升级为企业开发者，企业开发者不能再改为个人开发者之类的。

Tips：JavaScript中命名法则最好使用`驼峰法`，如`userActions`。而MySQL中不区分大小写，所以可以采用下划线命名法，如`user_actions`，在`SELECT`查询时使用`AS aliasName`设置别名即可。

缓存设计

缓存结构设计示例。

说明

常规结构：

```
Type:Key:SubKey
```

用 `:` 区分，在管理工具上会自动变成文件夹可收缩。

注册频率限制

Key：

```
reg:#手机唯一id或注册ip#
```

Value：

```
成功注册的用户名或手机号
```

（记录可以有迹可循，如果不需要查询，直接记录固定值，如 `1` 即可）

TTL： 1800(s)

说明：

> 注册成功后创建该 `key` ；判断，如果取到非 `null` 值，禁止注册。

登录尝试限制

Key：

```
trial:#手机唯一id或登录ip#
```

Value：

```
错误尝试次数
```

TTL： 1800(s)

说明：

> 初次尝试新建该 `key` ，值为1；判断，如果值大于3，禁止登录。

复杂缓存结构示例

接口性能监控，如图：

HASH: api:20150527 Size: 10

row	key	value
1	total	{"count":87278,"success":87139,"avg":8.01417501922212,"max":6687.243,"min":1.527}
2	device-cmic	{"count":12707,"success":12707,"avg":6.090618635397813,"max":6687.243,"min":3.469}
3	android-fcsr	{"count":4120,"success":4120,"avg":8.756312864077666,"max":241.061,"min":1.743}
4	android-familycamera	{"count":61414,"success":61361,"avg":6.876093658838649,"max":3481.942,"min":1.527}
5	/binding/notices	{"count":51580,"success":51579,"avg":5.965237092615151,"max":3481.942,"min":1.527}
6	/user/v5_login	{"count":3556,"success":3556,"avg":6.182495500562445,"max":241.061,"min":1.743}
7	/device/locale	{"count":312,"success":312,"avg":4.634733974358977,"max":10.962,"min":3.806}
8	/device/list	{"count":5765,"success":5764,"avg":33.379775329632295,"max":3119.883,"min":1.707}
9	/version/stable	{"count":261,"success":261,"avg":7.29767049808429,"max":21.781,"min":4.95}
10	/device/user_list	{"count":9563,"success":9563,"avg":6.374089825368599,"max":6687.243,"min":3.848}

结构：

Key：

```
api:#记录日期#
```

Value： 数组（通过Redis `HSet` 和 `HGet` 命令进行存储和读取）

```
[
  key：'#total（总计）/来源（如android-xxx/ios-xxx/device-xxx/web-xxx
  value: { // JSON.stringify(#对象#)
    count: '请求次数',
    success: '成功次数',
    avg: '平均响应时间',
    max: '最大响应时间',
    min: '最小响应时间'
  }
]
```

行为驱动开发实践

本实践项目源码：https://coding.net/u/willin/p/bdd-practice/git

配置数据库

开启MySQL和Redis服务。

创建数据库 `bdd` 。根据 `数据库设计` 章节创建 `user` 、 `usermeta` 两张表。

初始化项目

```
git init
npm init
```

安装ESLint和Babel环境

```
cnpm i --save-dev eslint babel-eslint eslint-config-airbnb eslint-p
cnpm i --save babel-register babel-runtime babel-plugin-transform-r
```



Tree：

```
├── .babelrc
├── .eslintignore
├── .eslintrc.json
├── .git
├── .gitignore
├── README.md
├── node_modules
└── package.json

2 directories, 6 files
```

参考： https://github.com/w2fs/best-practice

创建配置文件。

配置ava、nyc

```
npm install ava nyc --save-dev
./node_modules/.bin/ava --init
```

Package.json修改：

```
"scripts": {
  "test": "NODE_ENV=test ./node_modules/.bin/nyc --reporter=text -
},
"nyc": {
  "lines": 95,
  "functions": 90,
  "branches": 90,
  "check-coverage": true,
  "report-dir": "./.nyc_output",
  "exclude": [
    "node_modules",
    "test",
    "test{,-*}.js",
    "**/*.test.js",
    "**/__tests__/**"
  ]
},
"ava": {
  "files": [
    "test/*.js",
    "test/**/*.js",
    "!**/_*/*.js",
    "!**/_*.js"
  ],
  "require": [
    "babel-register"
  ],
  "babel": "inherit"
}
```

参考项目init代码： https://coding.net/u/willin/p/bdd-practice/git/tree/5c42541a2985b54619d09372ef05fc999b108f9a

用户登陆接口实现

设计

Route： `/user/login`

Payload：

```
{
  username: joi.alternatives().try(
    joi.string().email().max(32),
    joi.number().integer().min(10000000000).max(19999999999),
    joi.string().min(3).max(16)
  ).required().description('手机号，邮箱，或用户名'),
  password: joi.string().min(6).max(255).required().description('密
  guid: joi.string().required().default('').description('设备唯一识别
}
```

Result：

登陆成功：

```
{
  status: 1,
  data: {
    token: 'Access Token',
    expires: 3600 // Access Token有效期
  }
}
```

通用错误

```
{
  status: 0,
  err_code: 500,
  error_msg: 'Server Error'
}
```

编码

首先编写测试用例， `test/user/login.js` 。注意测试的顺序：

1. 200 登录成功
2. 400 参数错误
3. 401 用户名或密码错误，连续三次
4. 403 超出限制，正确用户密码登录

并且需要注意：

1. 测试前需要添加测试数据（测试用户），且信息不能与其他测试用例冲突（并行执行测试）
2. 测试后要删除测试数据，不要使用清空数据库之类的操作，以免对其他测试用例产生影响
3. 测试前也需要删除测试数据（以免前一次测试失败数据未删除而产生数据污染）

检查测试用例是否覆盖完整，以及测试用例是否写错。

这时候直接开始跑测试用例的话会报错。

测试用例参考： https://coding.net/u/willin/p/bdd-practice/git/blob/master/test/user/login.js

根据测试用例，开始编写功能模块代码。

另外，有一种情况是测试无法覆盖的，就是登录半小时的限制，我们也没有必要让测试用例一直运行等待半个小时再测。可以直接检查Redis里的缓存是否正常，以及TTL超时是否在合理范围内。

示例：

```
test('Login trial redis ttl', async(t) => {
  const value = await client.get('trial:guid-xxx');
  // 循环错误3次，加上已经限制还再继续尝试的1次
  t.is(value, 4);
  const ttl = await client.ttl('trial:guid-xxx');
  // 限制超时应当小于半小时
  t.true(ttl <= 1800);
});
```

剩下的编码部分就没什么可讲的了。 注意逻辑判断，测试代码覆盖率，没必要的判断不要加。

注意点：

- 数据库连接，使用连接池，并在所有查询完成后释放；
- 数据库查询禁止 `select field1, (select xxx) as field2` 嵌套查询；
- 慢SQL，如多张表 `JOIN` 的查询，根据业务逻辑，考虑加Redis缓存；
- 代码覆盖率要求 `95%` 以上，分支覆盖 `90%` 以上，只有异常捕获的代码和测试环境下的分支可以ignore；
- 不要用 `[].forEach()` 方法做轮询，直接用 `for` ；
- 算法、逻辑细节。

运维

常用工具

性能监控

开放接口文档

- NewRelic: https://rpm.newrelic.com/api/explore/applications/list
- 阿里云: https://help.aliyun.com/document_detail/28617.html

常用SHELL命令

推荐使用zsh： http://ohmyz.sh/

查看磁盘可用空间

```
df -h
```

> -h, --human-readable print sizes in human readable format (e.g., 1K 234M 2G)

```
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda4       321G  1.4G  304G   1% /
none            4.0K     0  4.0K   0% /sys/fs/cgroup
udev             16G  4.0K   16G   1% /dev
tmpfs           3.2G  968K  3.2G   1% /run
none            5.0M     0  5.0M   0% /run/lock
none             16G     0   16G   0% /run/shm
none            100M     0  100M   0% /run/user
/dev/sda2       185M   75M  102M  43% /boot
/dev/sda5        28G  916M   26G   4% /var
/dev/sda6       184G  1.7G  173G   1% /usr
/dev/sda7       234G  1.4G  221G   1% /home
/dev/sda1        94M  2.6M   91M   3% /boot/efi
```

查看内存使用

```
free -m
```

> -b,-k,-m,-g show output in bytes, KB, MB, or GB

```
                total        used        free      shared     buffers
Mem:            32105        1503       30602           0         223
-/+ buffers/cache:            648       31457
Swap:           61034           0       61034
```

性能分析

```
top
```

```
top - 11:13:36 up 12 days, 14:00,  3 users,  load average: 0.02, 0.
Tasks: 211 total,   1 running, 210 sleeping,   0 stopped,   0 zombi
Cpu(s):  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%
Mem:  32876316k total,  1535196k used, 31341120k free,   229204k bu
Swap: 62499836k total,        0k used, 62499836k free,   646796k ca

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAN
```

常用操作：

```
top   #//每隔5秒显式所有进程的资源占用情况
top -d 2  #//每隔2秒显式所有进程的资源占用情况
top -c  #//每隔5秒显式进程的资源占用情况，并显示进程的命令行参数(默认只有进程
top -p 12345 -p 6789  #//每隔5秒显示pid是12345和pid是6789的两个进程的资
top -d 2 -c -p 123456 #//每隔2秒显示pid是12345的进程的资源使用情况，并显
```

搜索文件

```
find . -name *.log  #//在当前目录下查找.log日志
```

服务器配置

创建用户

```
adduser xxx
# 输入密码
```

注：还有一个 `useradd` 命令，不会创建用户目录。

```
sudo vi /etc/sudoers
```

在

```
root    ALL=(ALL:ALL) ALL
```

后插入一行

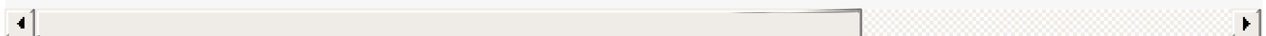
```
xxx(用户名)    ALL=(ALL:ALL) ALL
```

切换到用户安装环境

```
su username
```

安装 zsh

```
sudo apt-get update
sudo apt-get install zsh curl git
sh -c "$(curl -fsSL https://raw.github.com/robbyrussell/oh-my-zsh/r
```



配置ssh免密码登录

```
mkdir ~/.ssh
chmod 700 ~/.ssh
cd ~/.ssh
touch authorized_keys
chmod 644 authorized_keys
vi authorized_keys
```

插入你的 `ssh` 公钥。

```
# 不存在创建（注意是在本地，不是远程服务器）
ssh-keygen
# 一直按回车，结束
# 存在直接查看
cat ~/.ssh/id_rsa.pub
```

安装node

（示例，从官网获取最新版本源码编译安装）

```
cd ~ #注意安装目录，最好在用户目录下，其他系统目录可能会有权限问题
wget -c https://nodejs.org/dist/v6.4.0/node-v6.4.0.tar.gz
tar zxvf node-v6.4.0.tar.gz
cd node-v6.4.0/
./configure
make
sudo make install
# 安装成功测试
node -v
npm -v
```

源码编译安装Redis、OpenSSL等步骤基本相同。具体可以参考官方文档。

Nginx安装配置

Nginx 1.9.5 之后的版本支持了 `HTTP/2` ，同时，也取消了对 `SPDY` 的支持。

以 `HTTP/2` 模块支持安装为例。

```
wget -c http://nginx.org/download/nginx-1.11.3.tar.gz
tar zxvf nginx-1.11.3.tar.gz
cd nginx-1.11.3/
./configure --with-pcre --with-http_ssl_module --with-http_v2_modul
```

异常处理

1.没装PCRE

```
./configure: error: the HTTP rewrite module requires the PCRE libra
You can either disable the module by using --without-http_rewrite_m
option, or install the PCRE library into the system, or build the P
statically from the source with nginx by using --with-pcre=<path> o
```

ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre/

查找并下载最新版本PCRE源码，并解压

```
wget -c ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre/pcre-
tar zxvf pcre-8.38.tar.gz
cd pcre-8.38/
pwd
# /home/user/nginx-1.11.3/pcre-8.38
```

修改 `configure` 命令为：

```
./configure --with-pcre=/home/user/nginx-1.11.3/pcre-8.38 --with-ht
```

2.没装OpenSSL

```
./configure: error: SSL modules require the OpenSSL library.
You can either do not enable the modules, or install the OpenSSL li
into the system, or build the OpenSSL library statically from the s
with nginx by using --with-openssl=<path> option.
```

https://www.openssl.org/source/

查找并下载最新版本OpenSSL源码，并解压

```
wget -c https://www.openssl.org/source/openssl-1.0.2h.tar.gz
tar zxvf openssl-1.0.2h.tar.gz
cd openssl-1.0.2h
pwd
# /home/user/nginx-1.11.3/openssl-1.0.2h
```

注： `openssl-1.1.0-pre6` 版本经测试无法安装。

```
./configure --with-pcre=/home/user/nginx-1.11.3/pcre-8.38 --with-op
```

1. 没装zlib

```
./configure: error: the HTTP gzip module requires the zlib library.
You can either disable the module by using --without-http_gzip_modu
option, or install the zlib library into the system, or build the z
statically from the source with nginx by using --with-zlib=<path> o
```

http://www.zlib.net/

查找并下载最新版本Zlib源码，并解压

```
wget -c http://zlib.net/zlib-1.2.8.tar.gz
tar zxvf zlib-1.2.8.tar.gz
cd zlib-1.2.8
pwd
# /home/user/nginx-1.11.3/zlib-1.2.8
```

```
./configure --with-pcre=/home/user/nginx-1.11.3/pcre-8.38 --with-op
```

安装

```
make
sudo make install
```

配置Nginx

参考修改安装目录下的默认配置 `conf/nginx.conf` :

```
#user  nobody;
worker_processes auto;
worker_rlimit_nofile 100000;

#pid        logs/nginx.pid;

events {
    worker_connections 10240;
    multi_accept on;
    use epoll;
}

http {
    include       mime.types;
    default_type  application/octet-stream;

    sendfile        on;
    keepalive_timeout  65;

    # 从项目载入nginx配置
    # include /home/user/project/conf/nginx.conf;
}
```

HTTP/2项目配置：

```
server {
        listen 80;
        server_name example.com;
        add_header Strict-Transport-Security max-age=15768000;
        return 301 https://example.com$request_uri;
}

server {
        listen       443 ssl http2;
        server_name  example.com;

        # ssl_dhparam        /home/user/project/config/dhparam.pem;
        ssl_certificate      /home/user/project/config/ssl.crt;
        ssl_certificate_key  /home/user/project/config/ssl.key;

        ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
        ssl_ciphers ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES128-
        ssl_prefer_server_ciphers on;
        ssl_session_cache shared:SSL:10m;
        ssl_session_timeout 10m;
        add_header Strict-Transport-Security "max-age=31536000;";
        #add_header  X-Content-Type-Options "nosniff";
        #add_header X-Frame-Options DENY;

        access_log off;
        error_log /home/user/logs/example.com.error.log crit;

        location / {
                if ($http_user_agent = Mozilla/4.0){
                  return 503;
                }
                proxy_set_header X-Real-IP $remote_addr;
                proxy_set_header X-Forwarded-For $proxy_add_x_forwa
                proxy_set_header Host $host;
                proxy_set_header X-NginX-Proxy true;
                proxy_pass http://127.0.0.1:8888/;
                proxy_redirect off;
        }
}
```

配置开机自启动

文件位置： `/etc/rc.local`

示例：

```
#!/bin/sh -e
#
# rc.local
#
# This script is executed at the end of each multiuser runlevel.
# Make sure that the script will "exit 0" on success or any other
# value on error.
#
# In order to enable or disable this script just change the executi
# bits.
#
# By default this script does nothing.

/usr/bin/nginx &
sleep 1
sudo -u willin /project/path redis &
exit 0
```

Node.js 项目，使用 `PM2` 工具：

```
pm2 startup
```

根据界面提示，如：

```
[PM2] You have to run this command as root. Execute the following c
      sudo su -c "env PATH=$PATH:/usr/local/bin pm2 startup linux -
```

执行相应代码。

如果已经添加过开机自启动，更新自启项目：

```
pm2 dump
pm2 save
```

持续交付工作流

阿里云持续交付平台： https://crp.aliyun.com/

1.触发器任务

点击红色箭头所指圆圈位置设置触发器任务。

一般情况下，需要部署到产品环境是侦听 `Master` 分支，集成测试可以为其他开发分支。

本文示例以一套完整的自动化测试部署流程为例，选择了 `Master` 分支。

2.代码检出

1 代码检出 → 1 集成测试 → 0 stage3

活动信息 | 任务列表＋

活动名称 代码检出
自动触发 立即执行
自动完成
异常通知
指定操作人

代码更新

选择插件 云Code 代码插件
代码仓库 https://code.aliyun.com/weix/demo.git
侦听分支 master

这里的信息都是自动填入的，无需做更改。

3.集成测试

如果是简单的测试脚本，如单元测试，不需要数据库的。可以直接使用阿里云的编译测试功能，如下图所示：

自动化部署

帮助

1 代码检出 → 1 集成测试 → 0 stage3

活动信息 | 任务列表＋

活动名称 集成测试
自动触发 立即执行
自动完成
异常通知
指定操作人

编译/测试

选择语言 Node 6.1
构建环境 国内
运行命令 npm install
npm test
输出物路径 关

如果有专门的测试服务，可以用如下图所示方式进行测试：

1 代码检出 → 1 集成测试 → 0 stage3

活动信息 | 任务列表＋

活动名称 集成测试

☑ 自动触发 立即执行

☐ 自动完成

☐ 异常通知

☐ 指定操作人

部署

选择插件 Ali 手动部署插件

部署包来源 不自动上传

目标机器 10.218.128.145 多台机器以逗号分割

部署路径

部署命令

查看示例脚本

登录用户 root 机器授权

提示：CRP提供的测试环境是Ubuntu，未安装数据库，但据说可以自己安装，目前还没有尝试过。

注意点

自动完成

左侧活动信息中，【自动完成】选项，如果勾选，则测试通过就会自动进入下一步（如部署产品环境），否则会停在这里，需要手动触发，如下图所示：



表单项

目标机器

填入测试服务器ip。

部署路径

可以是用户目录，如 `/home/user/`

或是项目目录，如 `/home/user/project`

无太大影响，因为【部署命令】中可以使用 `cd` 命令。

一般这里我填入的是用户目录。

部署命令

流程：

1. 根据需要，启动、重启数据库/缓存服务（一般可以不用放在自动测试流程里）
2. 进入项目目录
3. 更新代码，新建当前版本分支，以备回滚操作
4. 更新依赖项
5. 启动测试脚本

Shell命令

```
cd /home/xxx-user/xxx-project/
git checkout .
git fetch
git checkout $CODE_VERSION
npm -d install
npm update
npm test
# 产品环境加入：
# pm2 reload xxx-server-name
```

登录用户

SSH 登入服务的用户名称

提示：系统需要您的目标机器添加部署公钥方可执行部署任务。请将公钥拷贝到服务器部署用户目录的$HOME/.ssh/authorized_keys文件中。

4.自动部署

新建流程

模板默认流程只有两个，需要新建的时候根据下图：



箭头所指小圆圈部分单击拖拽新建一个工作流，并将结束定向到新的工作流上。

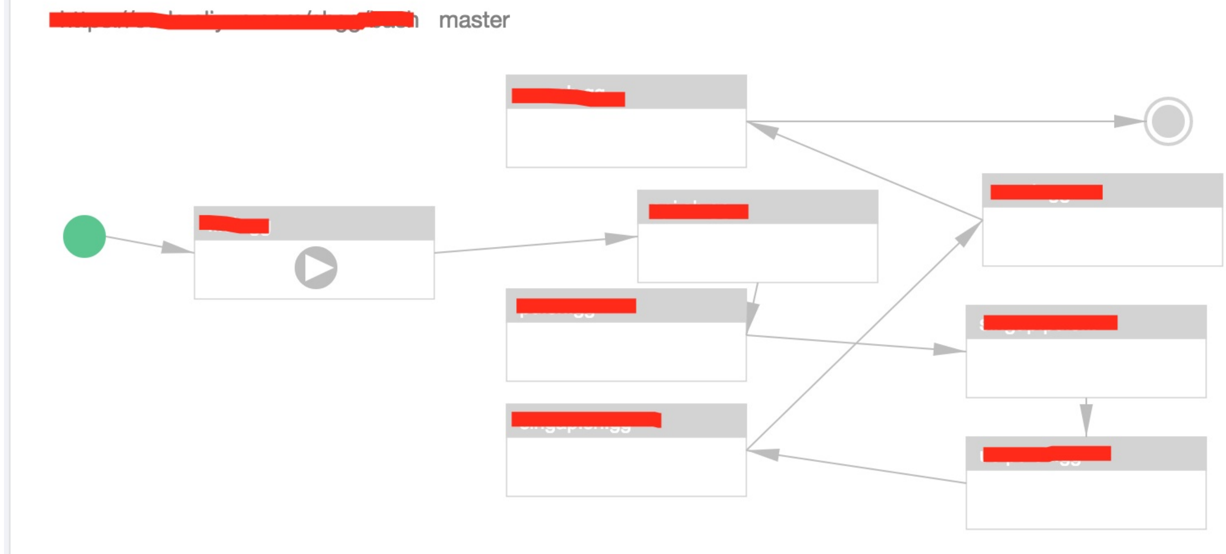
1 代码检出
1 集成测试
1 自动部署

活动信息
任务列表 +
活动名称 自动部署
自动触发 立即执行
自动完成
异常通知 邮件 请选择成员
指定操作人
部署
选择插件 Ali 手动部署插件
部署包来源 不自动上传
目标机器 10.218.128.145 多台机器以逗号分割
部署路径
部署命令
查看示例脚本
登录用户 root 机器授权

注意点

1. 【自动触发】、【自动完成】勾选上，如果需要，还可以打开【异常通知】
2. 【目标机器】如有多台负载均衡横向扩展的相同环境机器以逗号分隔

重启所有服务



虽然CRP中工作流可以一个点流出至多个点，但只有第一个子任务会执行。

所以我将自动重启任务改为了串行执行。

配置手动启动

如上图所示，将第一个代码检出任务的【自动触发】勾选去掉。

后续的每个工作流可以是每一台单独服务器或是每几台相同环境的负载均衡机器。

重启Shell脚本

```
pm2 kill
rm -f /home/xxx-user/xxx-project1/logs/*.log
rm -f /home/xxx-user/xxx-project2/logs/*.log
pm2 start /home/xxx-user/xxx-project1/app.json
pm2 start /home/xxx-user/xxx-project2/app.json
```

版本回退

历史记录

#138	2016-08-03 15:29:36 fc8ca759 回滚至此版本	完成
#137	2016-08-03 08:57:47 e2ae5233 回滚至此版本	完成
#136	2016-08-02 16:37:40 bfb497a2	失败
#135	2016-08-01 15:24:29 b209dca9 回滚至此版本	完成
#134	2016-08-01 11:59:23 a743d25d 回滚至此版本	完成
#133	2016-08-01 11:45:16 a4b49095 回滚至此版本	完成
#132	2016-07-29 15:34:17 0b5555b1 回滚至此版本	完成
#131	2016-07-28 15:47:55 3a5bbe8f 回滚至此版本	完成
#130	2016-07-28 10:11:18 12eac5f4 回滚至此版本	完成
#129	2016-07-28 09:57:37 502a454f 回滚至此版本	完成

< 1 2 3 4 5 >

如果版本部署失败，可以回滚至之前任意成功部署版本。